Complete and Infinite Traces: A descriptive model of computing agents

Kevin S. Van Horn

Computer Science Department
California Institute of Technology

5207:TR:86

Complete and Infinite Traces:
A descriptive model of computing agents

Kevin S. Van Horn

Computer Science Department
California Institute of Technology

5207:TR:86

The research described in this paper was sponsored by
the Defense Advanced Research Projects Agency, ARPA Order No. 3771,
and monitored by the Office of Naval Research
under contract number N00014-79-C-0597

© California Institute of Technology, 1986

COMPLETE AND INFINITE TRACES:
A descriptive model of computing agents

Kevin S. Van Horn
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Abstract: A model of computing agents is presented. Computing agents are modeled as *processes*, which are essentially sets of traces representing possible complete sequences of actions performed by an agent and its environment. Some technical difficulties with infinite traces are resolved, with the result that one may take the parallel composition of any countable set of processes, after possibly renaming some symbols.

0. Introduction

One indication of how well we understand some phenomenon is our ability to provide an adequate mathematical model of it. Such a model provides a firm basis for reasoning about the phenomenon; in its absence we are vulnerable to the treacheries of informal reasoning, and we are severely limited in our ability to analyze complex instances of the phenomenon. Computer science, as distinguished from the natural sciences, is the study of phenomena of our own invention; alas, invention does not necessarily entail comprehension. Adequate mathematical models are just as indispensable for computer science as for other disciplines. In this paper we propose a general model, *CIT* (Complete and Infinite Traces), of a "phenomenon" of great importance in computer science: computing agents.

The motivation for developing as general a model as possible is a desire for conceptual economy. For example, many different approaches to concurrent programming have been investigated, and many proposals have been put forth for the semantics of various concurrent languages; wouldn't it be nice if all of these could be described and reasoned about within the same underlying mathematical framework? This would facilitate the introduction of useful new programming constructs to a language without wreaking havoc with the semantics. As an example, there have been a number of proposals for the denotational semantics of CSP [7][1][2][3], and it is not obvious how they should be extended to include Alain Martin's probe function [4]. In addition, the ability to describe hardware systems

within this same framework would help to further erode the dichotomy between hardware and software and aid us in applying the same reasoning and design methods to both kinds of systems.

What is a computing agent? In one view, a computing agent is something that takes an input and produces as output some function of it. This is the view taken in the traditional denotational semantics of sequential programming languages. An imperative program is considered to denote a function from an initial to a final state, and a functional program is considered to denote a function on some domain. Such a view is clearly inadequate for describing hardware devices other than purely combinational logic, and is too simplistic even for describing most programs—except in scientific computing, few useful programs are of the "give me an input and I'll give you an output" variety.

The view we take in this paper is this: computing agents are objects which may perform various *actions*, thus exhibiting some discrete *behavior*, and this behavior may be influenced by the actions of other agents. In the case of a digital circuit the relevant actions are voltage transitions. A computer program may perform such actions as assigning a value to a variable, initiating a CSP-style communication action, or writing a character to the user's screen, and may respond to actions such as the user hitting a key on the keyboard.

This is a different view of actions (sometimes referred to as events) than is taken in models such as Milner's CCS [5] or Hoare, Brookes, and Roscoe's failures model of CSP [2], in which actions require the participation of more than one agent. Here we consider every action to be performed by some single agent. A CSP communications action is viewed as two actions, one performed by the sender and the other by the receiver.

An action performed by the computing agent itself is an output action, while an action performed by some exterior agent is an input action. An output action is not necessarily "seen" by any external agent; it may be an internal action. There is an essential asymmetry between input actions and output actions: the computing agent can control which output actions take place, but has no control over input actions. This asymmetry is the reason for using the term "computing *agent*" instead of "computing *system*." We wish to describe the behavior of some object which will be made to interact with other objects, without knowing *a priori* anything about the behavior of those other objects.

1. Traces

We shall use *traces* to represent sequences of actions. It is assumed that the reader has had some exposure to formal language theory; below we present the definitions and notations which will be used in this paper.

A *trace* is a sequence of symbols taken from some *alphabet*, which is just a set of symbols. The empty trace is denoted by ε, and the trace of unit length formed from the symbol x is denoted $x\cdot$. The set of all finite traces formed from alphabet A is written A^*, the set of all infinite traces formed from A is written A^ω, and A^∞ is just $A^* \cup A^\omega$.

The catenation of two traces t and u is written as the juxtaposition of the two, i.e. tu.

If t is an infinite trace then tu is undefined.

The length of a trace t is written as $\ell(t)$; if t is infinite, then $\ell(t) = \omega$. We write $\Phi(t)$ for "t is finite." We shall often omit the phrase $\Phi(t)$ when it is obvious from context.

A *prefix* of a trace is any finite initial subsequence of it. We write $t \leq t'$ to mean "t is a prefix of t'", i.e. $\Phi(t) \wedge (\exists u :: t' = tu)$. Note that $\leq$ is a partial order on traces (except that $t \leq t$ doesn't hold if t is infinite).

The set of prefixes of a trace t is $pref\, t = \{ t' \mid t' \leq t \}$. Note that $pref\, t$ contains only finite traces. For a set of traces T we similarly define $pref\, T = \bigcup(t : t \in T : pref\, t)$. We say that T is *prefix-closed* if $pref\, T \subseteq T$.

A *chain* of traces is a set of finite traces S s.t. $t \leq t'$ or $t' \leq t$ for all $t, t' \in S$. Every chain of traces S has a least upper bound, which we shall denote $\mathcal{L}(S)$; if S is finite then $\mathcal{L}(S)$ is just the maximum element of S, otherwise it is the unique infinite trace t s.t. $pref\, t = pref\, S$. Note that $pref\, t$ is a chain for all traces t, and that $t = \mathcal{L}(pref\, t)$.

Given a trace t and trace set T, we define $next(t, T)$ to be $\{ a \mid ta \cdot \in pref\, T \}$.

We write $t\#A$ for the number of occurrences of symbols from A in t, and $t\#a$ for $t\#\{a\}$ if a is a symbol.

2. Sequential Processes

The mathematical structure which we will use to model a computing agent is called a *process*. A process is essentially a set of traces plus an input and an output alphabet. Each symbol in the output alphabet represents some action which the agent may perform, and each symbol in the input alphabet represents some external action which may affect the agent. We consider all actions to be instantaneous, having no duration in time; if an "action" actually does have some nonzero duration, we will model it by two actions, one which indicates initiation and another which indicates termination.

We assume that there is some "starting point", some point in time before which neither the computing agent nor the agents interacting with it perform any action. We can then imagine an observer who is present from the starting point until the end of eternity, and who records the sequence of actions (both input and output) which are performed. This sequence we call a *complete history*. If we imagine that our observer gets bored after a finite amount of time and wanders off, the sequence he records we call a *partial history*, and will be a prefix of some complete history. Every complete history which might be recorded by our imaginary observer must be in the trace set of the process describing the computing agent in question.

By using traces this way we implicitly assume that no two distinct actions may occur at *exactly* the same time; one always precedes the other, even if the time separating them is too small to be measured. This is justifiable since, for the systems we will consider, only a finite number of actions may occur within a finite interval of time, and this finite interval contains an uncountably infinite number of points.

The process modeling a computing agent must include in its trace set any sequence of actions which we cannot guarantee will *not* occur, given our knowledge of the computing agent. Thus it could happen that, upon further investigation, we will find that some element of the trace set is not a possible complete history at all. For example, the trace set may indicate that two concurrent actions a and b may occur in either order, whereas the actual operation of the agent may be such that a will always occur before b. If the trace set contains elements which are not in reality possible complete histories, our model may be *inadequate*, but as long as the trace set contains every possible complete history, we will consider the process to be a *correct* model of the computing agent.

It is evident that not all sets of traces are valid representations of the behavior of some computing agent. For example, suppose that t is some possible partial history; furthermore, suppose that when the sequence of actions t has occurred the next action that occurs must be an input action, i.e. for all a s.t. $ta\cdot$ is a partial history, a is an input action. Then t is a possible complete history, since the required input action may never occur, and so t must be in the trace set. In addition, as is often the case, strange things happen at infinity. There are mathematical difficulties with parallel composition when infinite traces are allowed. There are also conceptual difficulties—if an infinite sequence of actions can never be observed in its entirety, what does an infinite trace mean? Obviously, an infinite trace must represent some sort of limit of finite behavior. The question then becomes, when should the l.u.b. of an infinite chain of partial histories be included in the trace set, and when should it not?

In order to simplify the task of characterizing the valid sets of traces, we assume that any process is either a sequential process or may be formed from the parallel composition of a number of sequential processes, where a sequential process is one which may model an agent that performs its output actions one at a time. Note that the input actions of such an agent may occur concurrently with each other and with the output actions.

In this section we will examine sequential processes. For the rest of this section, I and O will be resp. the input and output alphabets of the sequential process in question, and we will write A for $I \cup O$.

If T is the trace set of a process, and t is a partial history, then $next(t,T)$ (see section 1) is the set of all possible next actions after the sequence t has occurred. Since the agent has no control over its input actions, any input action may occur at any time. Hence we give the following requirement on the trace set T of a sequential process:

$R1$: $$I \subseteq next(t,T) \text{ for all } t \in pref\ T$$

A sequential computing agent is excited or *enabled* at some point in time if it is about to perform an output action, and we can guarantee that if no input action interferes, some output action will occur. If t is the sequence of actions which have occurred up to this point in time, we say that the agent is enabled at t. For example, an inverter with a 0 value at both input and output is enabled: it is about to perform a transition from 0 to 1

on the output, and is guaranteed to do so unless there is a transition from 0 to 1 on the input line first. Note that t cannot be a complete history if the agent is enabled at t.

The operation of a sequential agent may be viewed as follows. At any point in time, either the agent is not enabled and it may be that no more actions will ever be performed, or the agent is enabled and will eventually either be disenabled by some input action or nondeterministically choose one of its possible next output actions and perform it. Note that this is not in general an appropriate view of the operation of an agent which comprises several concurrently operating components. With these ideas in mind we can now say what the infinite complete histories should be for a sequential agent. Given any prefix-closed chain S of partial histories, $\mathcal{L}(S)$ is a complete history *provided that* whenever $s \in S$ and the agent is enabled at s, there is some $su \in S$ s.t. either the agent is not enabled at su or $u\#O > 0$. Note that if S is finite then this says that $\mathcal{L}(S)$ is a complete history iff the agent is not enabled at $\mathcal{L}(S)$.

We now formalize these ideas as a requirement for sequential processes.

Definition: Given $T \subseteq A^\infty$ and $t \in \mathit{pref}\, T$,

$$en_O(t,T) \iff t \notin T \wedge \mathit{next}(t,T) \cap O \neq \emptyset$$

When there is no ambiguity we will simply write $en(t,T)$. If T is the trace set of a sequential process then $en(t,T)$ is true if the agent modeled is enabled at t.

Definition: We say that S is a *proper chain* of $T \subseteq A^\infty$, denoted $S\ \mathit{chainof}_O\ T$, iff S is a prefix-closed chain and $S \subseteq \mathit{pref}\, T$ and

$$\forall s : s \in S \wedge en(s,T) : (\exists u : su \in S : \neg\, en(su,T) \vee u\#O > 0)$$

When there is no ambiguity we will simply write $S\ \mathit{chainof}\ T$. A proper chain is intended to be the prefix set of a possible complete history.

Definition: Given $T \subseteq A^\infty$, the *completion* of T is

$$T^{cp} = \{\, \mathcal{L}(S) \mid S\ \mathit{chainof}\ T \,\}$$

It is easily shown that $\mathit{pref}\, T = \mathit{pref}\, T^{cp}$ and $(T^{cp})^{cp} = T^{cp}$ and $T \cap A^* \subseteq T^{cp} \cap A^*$. If $T = T^{cp}$ we say that T is *complete*. Note that there may be some infinite traces of T which are not in T^{cp}. We then require that the trace set of a sequential process be complete:

$R2$: $$T = T^{cp}$$

As we will see in the next section, requirements $R1$ and $R2$ on sequential processes relieve the difficulties with parallel composition when infinite traces are allowed.

Note. This completeness requirement is analogous to Soundararajan's completeness [3] and Back's *closedness* [8] requirements, with the difference that, due to the input-output dichotomy, we do not take the limit of every chain of partial histories.

We may wish to say that the future behavior of an agent after a certain sequence of actions t has occurred is utterly unknown. There may be several reasons for this. It may be that the occurrence of an input action at the wrong time may cause something disastrous to happen, such as a flip-flop going metastable (in which case the behavior may no longer be considered discrete), or a dazzling display of pyrotechnics if we have a circuit designed by a member of the Screenwriter's Guild. It may be that we are truly ignorant of what the future behavior might be after the sequence t has occurred. Or it may be that we do not wish to consider what the future behavior might be. In addition, having an explicit notion of "utterly unknown" or "undefined" is useful in applying this theory to recursive programming language semantics (a topic to be covered in a future paper.) If the future behavior of a sequential agent is unknown after t has occurred, we say that the agent (resp. the process modeling it) is *broken* at t.

So in addition to T, which gives the possible complete histories for our computing agent, we have a set $U \subseteq T$ giving the traces at which the agent is broken. We call U the *breakage set.* We place some restrictions on what U can be. First of all, if the agent is broken at t then after t anything may happen, thus $tu \in T$ for all $u \in A^\infty$, and since the agent stays broken once it breaks, $tu \in U$ also. Secondly, since the agent may be broken at t only if t is finite, the only infinite traces in U should be those required by the previous rule.

Definition: Given $U \subseteq A^\infty$, the *convex closure* of U is

$$U^{cv} = \{\, tu \in A^\infty \mid t \in U \wedge \Phi(t) \,\}$$

Note that $(U^{cv})^{cv} = U^{cv}$. If $U = U^{cv}$ we say that U is *convex.* Note that $U = U^{cv}$ is equivalent to

$$(\forall t, u : t \in U \wedge \Phi(t) \wedge u \in A^\infty : tu \in U) \wedge (\forall t : t \in U : \exists t' : t' \leq t : t' \in U)$$

We then require that U, the set of traces at which the process is broken, be convex:

$R3$: $$U = U^{cv}$$

Note. This convexity requirement is similar to Soundararajan's convexity [3] and Back's *flatness* [8] requirements.

Note that requirement $R1$ made the introduction of breakage sets into our model necessary. In other models, such as trace theory [9][10], no breakage set is required because processes are not required to satisfy $R1$. In these models $R1$ is not used either because

there is no distinction between input and output actions or because the trace set is regarded as a *specification* of the allowed behavior of both the agent and its environment. In the latter case there are consistency or "composability" requirements which must be satisfied to allow the parallel composition of two processes. This is undesirable for a descriptive model; we would like to be able to do the parallel composition of *any* set of processes with disjoint output alphabets. As we shall see, requirements $R1$ and $R2$ on sequential processes are sufficient to allow this.

With these conditions we can now give the formal definition of a sequential process.

Definition: A *sequential process* is a tuple $S = \langle I, O, T, U \rangle$ such that

a. I and $O \neq \emptyset$ are disjoint alphabets called the input and output alphabets of S respectively.

b. $U \subseteq T \subseteq (I \cup O)^\infty$ and $T \neq \emptyset$.

c. $R1$ holds: $I \subseteq next(t, T)$ for all $t \in pref\ T$.

d. $R2$ holds: $T = T^{cp}$.

e. $R3$ holds: $U = U^{cv}$

For such an S we define

$$\mathbf{a}S = I \cup O \qquad \mathbf{i}S = I \qquad \mathbf{o}S = O$$
$$\mathbf{t}S = T \qquad \mathbf{u}S = U \qquad \mathbf{p}S = pref\ T$$

Example 2.1: A C-element is a digital circuit with two inputs and one output. A "transition" refers to a change from high to low or from low to high voltage on a wire. A C-element waits for transitions to occur at both of its inputs, and then performs a transition at its output. If two input transitions occur without an output transition separating them, we can't say just what will happen. Letting a and b represent transitions on the inputs and c a transition on the output, we model a C-element as the sequential process

$$\langle \{a, b\}, \{c\}, T \cup U, U \rangle \qquad \text{where}$$
$$T = S^{*\underline{\ }}\{a\cdot, b\cdot, \varepsilon\} \cup S^{\underline{\omega}}$$
$$S = \{a\cdot b\cdot c\cdot, b\cdot a\cdot c\cdot\}$$
$$U = \{\, tx\cdot \mid x \in \{a, b\} \wedge t \in pref\ T \wedge t\#x > t\#c \,\}^{cv}$$

($S^{*\underline{\ }}$ is the set of traces formed by catenating together any finite sequence of elements from S, and $S^{\underline{\omega}}$ is the set of traces formed by catenating any infinite sequence of elements from S. For two sets of traces R_1 and R_2, $R_1 R_2$ is $\{\, tu \mid t \in R_1 \wedge u \in R_2 \,\}$.)

Example 2.2: Let S model a set-reset flip-flop, where both the inputs and the output are initially low. When the flip-flop is set, the the set signal must not be removed until the output is high, and when it is reset, the reset signal must not be removed until the output is low, otherwise the flip-flop may go metastable. In addition, the set and reset

signals should never be simultaneously high. Let r, s, and q represent transitions on the reset line, set line, and output respectively. Defining $A = \{r, s, q\}$, we can define S as

$$\langle \{s,r\}, \{q\}, T \cup U, U \rangle \qquad \text{where}$$
$$T = \{\, t \in A^\infty \mid \forall v : v \leq t : (h(r,v) \wedge h(q,v) \vee h(s,v) \wedge l(q,v)) \Leftrightarrow vq\cdot \leq t \,\}$$
$$U = (U_1 \cup U_2)^{cv}$$
$$U_1 = \{\, tr\cdot \mid t \in pref\, T \wedge (h(r,t) \wedge h(q,t) \vee l(r,t) \wedge h(s,t)) \,\}$$
$$U_2 = \{\, ts\cdot \mid t \in pref\, T \wedge (h(s,t) \wedge l(q,t) \vee l(s,t) \wedge h(r,t)) \,\}$$
$$h(x,t) \iff odd(t\#x)$$
$$l(x,t) \iff \neg h(x,t)$$

Example 2.3: Suppose we have a coroutine which, in a neverending cycle, waits to be passed an integer x, then passes back $x * n$ where n is the number of times it has been called. We model this as the sequential process

$$\langle I, O, T \cup U, U \rangle \qquad \text{where}$$
$$I = \{\, \langle i, x \rangle \mid x \text{ is an integer} \,\}$$
$$O = \{\, \langle o, x \rangle \mid x \text{ is an integer} \,\}$$
$$A = I \cup O$$
$$T = \{\, t \in A^\infty \mid \forall v, x : v \leq t : v\#O \leq v\#I \wedge (v\langle i,x\rangle\cdot \leq t \Rightarrow v\langle i,x\rangle\cdot\langle o, x * (v\#I)\rangle\cdot \leq t) \,\}$$
$$U = \{\, ty\cdot \mid y \in I \wedge t \in pref\, T \wedge t\#I > t\#O \,\}^{cv}$$

3. Parallel Composition and General Processes

We now turn our attention to parallel composition. We define parallel composition in a manner analogous to that used in trace theory, using the *projection* operator. The projection of a trace t onto an alphabet A, written $t\lceil A$, is just t with all symbols not in A removed. For finite traces we define it as follows:

$$\varepsilon\lceil A = \varepsilon$$
$$(a\cdot u)\lceil A = a\cdot(u\lceil A) \text{ if } a \in A$$
$$(a\cdot u)\lceil A = u\lceil A \text{ if } a \notin A$$

If T is a set of traces then $T\lceil A = \{\, t\lceil A \mid t \in T \,\}$. Noting that

$$(pref(t\lceil A) = pref(t)\lceil A) \quad \text{and} \quad (t \leq t' \Rightarrow t\lceil A \leq t'\lceil A)$$

for all finite traces t and t', we see that $t\lceil A = \mathcal{L}(pref(t)\lceil A)$ for all finite t. So for infinite traces t we define

$$t\lceil A = \mathcal{L}(pref(t)\lceil A)$$

The parallel composition of a set of processes models the computing agent that is the aggregate of the computing agents modeled by the elements of the set. It is meaningful to form the parallel composition of a set of processes as long as the set is *composable*, which for a set of sequential processes K means that K is countable and

$$(\forall S, S' : S, S' \in K : \mathbf{o}S \cap \mathbf{o}S' = \emptyset \vee S = S')$$

This just says that we have been consistent in assigning symbols to represent actions, and have not used one symbol to represent two kinds of actions performed by distinct computing agents. Note that K may be an infinite set. The reason for this is that in some concurrent languages processes may be created "on the fly." We can model this by assuming that all processes which might be created already exist, but are quiescent until "awakened" by the parent process. Since we may not wish to put an upper bound on the number of such processes, it is convenient to pretend we have an infinite number of them, of which all but a finite number are quiescent at any moment, just as we often pretend that a memory allocator has infinite memory resources to draw upon.

A moment's reflection will reveal that a sequence of actions is a possible complete history for the aggregate of a collection of computing agents if and only if for any of the agents, upon projecting the sequence onto the set of actions which may be seen or performed by that agent we get a possible complete history for that agent. Hence we define the parallel composition of a set of sequential processes as follows.

Definition: Given a composable set K of sequential processes, the parallel composition of K is

$$\begin{aligned}(\|K) = &\langle I, O, T, U\rangle \qquad \text{where}\\ &O = \bigcup(S : S \in K : \mathbf{o}S)\\ &I = \bigcup(S : S \in K : \mathbf{i}S) - O\\ &T = \{\, t \in (I \cup O)^{\infty} \mid \forall S : S \in K : t\lceil \mathbf{a}S \in \mathbf{t}S \,\}\\ &U = \{\, t \in T \mid \exists S : S \in K : t\lceil \mathbf{a}S \in \mathbf{u}S \,\}\end{aligned}$$

Using the terminology of trace theory, we will often call the parallel composition of a set of processes the *weave* of the processes. We then define a process to be anything that is the weave of a composable set of sequential processes.

Definition: A *process* is a member of the class

$$P = \{\, \|K \mid K \subseteq S \text{ and } K \text{ is composable} \,\}$$

where S is the class of sequential processes. If P is a process, then $\mathbf{a}P$, $\mathbf{t}P$, etc. are defined just as for a sequential process. Note that only sequential processes are required to satisfy $R2$ and $R3$, and processes in general may not satisfy $R2$ or $R3$.

For any sequential process S, $\{S\}$ is composable and $\|\{S\} = S$, and so $\mathcal{S} \subseteq \mathcal{P}$. We define composability and the weave for general subsets of $\mathcal{P}$ just the same as for subsets of $\mathcal{S}$. The question then arises as to whether the weave of a composable set of processes is always a process. The answer is affirmative. We prove this beginning with the following lemma.

Lemma 3.1: *Let J be a countable set of countable subsets of $\mathcal{P}$ such that*

$$\forall K, K', P, P' : K, K' \in J \wedge P \in K \wedge P' \in K' : \mathbf{o}P \cap \mathbf{o}P' = \emptyset \vee K = K' \wedge P = P'$$

Then each $K \in J$, as well as $\bigcup J$ and $\{ \|K \mid K \in J \}$, is composable, and

$$\Big\| \{ \|K \mid K \in J \} = \Big\| \bigcup J$$

Proof: Simple application of the definitions. □

Note that if we define the binary weave operator by $P \| Q \stackrel{\text{def}}{=} \big\| \{P, Q\}$ then binary weave is commutative, and as a consequence of Lemma 3.1 it is also associative.

Theorem 3.2: *If C is a composable subset of $\mathcal{P}$ then $\|C \in \mathcal{P}$.*

Proof: For each $P \in C$ there is some composable set $K_P \subseteq \mathcal{S}$ such that $P = \|K_P$. Let $J = \{ K_P \mid P \in C \}$. It is easily seen that J satisfies the requirements of the previous theorem, hence

$$\Big\| \bigcup J = \Big\| \{ \|K \mid K \in J \} = \Big\| \{ \|K_P \mid P \in C \} = \Big\| C$$

Then since $\bigcup B \subseteq \mathcal{S}$ we have that $\|C \in \mathcal{P}$. □

One of the problems in developing a trace-based theory of computing agents, which we alluded to earlier, is ensuring that the weave always produces meaningful results. Suppose we have a composable set $K \subseteq \mathcal{S}$ and a trace $t \in (\bigcup_{S \in K} \mathbf{a}S)^*$ such that $t \lceil \mathbf{a}S \in \mathbf{p}S$ for all $S \in K$. Since $t \lceil \mathbf{a}S$ is a possible partial history of the agent modeled by S for all $S \in K$, we should expect t to be a possible partial history of the aggregate of the agents modeled by processes in K, i.e. we expect that $t \in \mathbf{p}(\|K)$. If we place no restriction on the trace sets of the elements of K, this will not in general be true. If our trace sets contain only finite elements, it is not hard to find obvious conditions on the trace sets to make this hold true. The importance of condition $R2$ (combined with $R1$) for sequential processes is that it makes this hold true even if there are infinite traces in the trace set.

Theorem 3.3: *For all composable sets $K \subseteq \mathcal{S}$,*

$$\mathbf{p}(\|K) = \{ t \in (\bigcup_{S \in K} \mathbf{a}S)^* \mid \forall S : S \in K : t \lceil \mathbf{a}S \in \mathbf{p}S \}$$

Proof: Let T be the right-hand side of the above equality. It is obvious that $\mathbf{a}(\|K) = \bigcup_{S \in K} \mathbf{a}S$, and that $\mathbf{p}(\|K) \subseteq T$. It remains to show that every element of T is an element of $\mathbf{p}(\|K)$. Given some $t \in T$, we show that $t \in \mathbf{p}(\|K)$ by constructing a $t' \in \mathbf{t}(\|K)$ such that $t \leq t'$, as follows:

First, for all natural numbers i we define the sequence $\sigma_i = 0 \cdot 1 \cdot \cdots i \cdot$, and we define the sequence $\tau = \sigma_0 \sigma_1 \sigma_2 \cdots$. The jth element of τ is denoted $\tau[j]$. The sequence τ has the important property that for all j and n there is some $k > j$ such that $\tau[k] = n$.

Enumerating the elements of K as S_0, S_1, ..., we define a chain of traces $\{t_j\}_j$ by

a. $t_0 = t$

b. Given t_j, let $i = \tau[j]$. If $i \geq |K|$ or $N \stackrel{\text{def}}{=} next(t_j \lceil \mathbf{a}S_i, \mathbf{t}S_i) \cap \mathbf{o}S_i = \emptyset$ then $t_{j+1} = t_j$. Otherwise $t_{j+1} = t_j a \cdot$ where a is some element of N.

In other words, for each j, if $i \stackrel{\text{def}}{=} \tau[j] < |K|$ we look at S_i and extend t_j by some action from $\mathbf{o}S_i$ if possible to produce t_{j+1}. $R1$ ensures that $t_{j+1} \lceil \mathbf{a}S_k \in \mathbf{p}S_k$ for all k, since $\mathbf{o}S_i$ and $\mathbf{o}S_j$ are disjoint for all $j \neq i$. We then see that $pref\{t_j \lceil \mathbf{a}S_i\}_j \subseteq \mathbf{p}S_i$ for all i. For all $s \in pref\{t_j \lceil \mathbf{a}S_i\}_j$ there is some u s.t.

$$(su \in pref\{t_j \lceil \mathbf{a}S_i\}_j) \wedge (next(su, \mathbf{t}S_i) \cap \mathbf{o}S_i = \emptyset \vee u \# \mathbf{o}S_i > 0),$$

(due to the form of τ) and hence $pref\{t_j \lceil \mathbf{a}S_i\}_j$ *chainof* $\mathbf{t}S_i$. Thus $t'_i = \mathcal{L}\{t_j \lceil \mathbf{a}S_i\}_j \in \mathbf{t}S_i$ for all i, due to $R2$. Then if $t' = \mathcal{L}\{t_j\}_j$ we have that $t \leq t'$ and $t' \lceil \mathbf{a}S_i = t'_i \in \mathbf{t}S_i$ for all i, and hence $t' \in \|K$. □

Theorem 3.3a: *(Corollary). For all composable sets $K \subseteq \mathcal{P}$,*

$$\mathbf{p}(\|K) = \{ t \in (\bigcup_{P \in K} \mathbf{a}P)^* \mid \forall P : P \in K : t \lceil \mathbf{a}P \in \mathbf{p}P \}$$

Proof: Follows from Lemma 3.1 and Theorem 3.3. □

A consequence of the above theorem is that $\mathbf{t}P$ is nonempty for all $P \in \mathcal{P}$.

4. Comparison with Trace Theory and Directions for Further Research

There are a number of differences between CIT and trace theory [9][10] which should be pointed out. Whereas CIT is intended to be used to describe and aid in reasoning about the behavior of a computing agent, trace theory is generally used as a means of *specifying* acceptable behavior of a component of a computing system. In trace theory a component is modeled by a trace structure, which is an alphabet plus a prefix-closed set of finite traces which specifies the acceptable partial histories of the interaction of a component

and its environment. A trace structure is then a specification of both a component and its environment, whereas a CIT process is a description of the behavior of a computing agent which makes no assumptions about the environment in which the agent will operate. Another difference is that trace theory deals with partial histories and thus specifies only "safeness" or invariance properties, whereas liveness properties may be discussed in the framework of CIT due to the use of complete traces. As an example, suppose that for some process P we wish to say that if condition ϕ holds and continues to hold long enough, then eventually condition ψ will holds. We can write this as

$$\forall t, s : s \leq t \in \mathbf{t}P \wedge \phi(s) : (\exists s' : s \leq s' \leq t : \neg\phi(s') \vee \psi(s'))$$

In general, how do we specify properties that we wish a computing agent to have? We suggest that the only properties of a process P which are really interesting for the purposes of verification are properties of the form

$$(\forall t : t \in \mathbf{u}P : \exists t' : t' \leq t : \psi(t'))$$

and

$$(\forall t : t \in \mathbf{t}P : \phi(t))$$

This suggests that perhaps some form of linear-time temporal logic [6] would be useful in specifying and reasoning about CIT processes.

Another interesting area of research is the application of CIT to the semantics of concurrent programming languages. Work is proceeding on the CIT-theoretic denotational semantics of concurrent languages. This has required defining an appropriate approximation ordering on sequential processes to allow recursive definitions. The denotational semantics will then be used as a foundation from which to develop axiomatic semantics and proof systems for these languages.

References

1. N. Francez, D. Lehmann and A. Pneuli. A Linear-History Semantics for Languages for Distributed Programming. *Theoretical Computer Science* 32 (1984), 25–46.

2. S. D. Brookes, C. A. R. Hoare and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journ. ACM* 31, no. 3 (July 1984), 560–599.

3. N. Soundararajan. Denotational Semantics of CSP. *Theoretical Computer Science* 33 (1984), 279–304.

4. A. J. Martin. The Probe: An Addition to Communication Primitives. *Information Processing Letters* 20, no. 1 (Jan. 1985), 125–130.

5. R. Milner. *Lecture Notes in CS, vol. 92: A Calculus of Communicating Systems*, Springer-Verlag, 1980.

6. Z. Manna and A. Pneuli. Verification of Concurrent Programs: the Temporal Framework. *The Correctness Problem in Computer Science* (R. S. Boyer and J. S. Moore, eds.), Academic Press, 1982, 215–273.

7. C. A. R. Hoare. Communicating Sequential Processes. *Comm. ACM* 21, no. 8 (Aug. 1978), 666–677.

8. R. Back. Semantics of Unbounded Nondeterminism. *Lecture Notes in CS, vol. 85: Automata, Languages and Programming. Proceedings, 1980* (J. de Bakker and J. van Leeuwen, eds.), Springer-Verlag, 1980, 51–63.

9. J. L. A. van de Snepscheut. *Lecture Notes in CS, vol. 200: Trace Theory and VLSI*, Springer-Verlag, 1985.

10. M. Rem. Concurrent Computations and VLSI Circuits. *Control Flow and Data Flow: Concepts of Distributed Programming* (M. Broy, ed.), Springer-Verlag, 1985, 399–437.